CIS 624

Spring 1998

Midterm (100 points)

Name: ....

1	Hello World	8 pts	
2	Interpreters	25 pts	
3	Closures	25 pts	
4	CPS	30 pts	
5	Types	12 pts	
6	Extra Credit	10 pts	

Hello World (8 points)

Write a Scheme function that appends two lists and convert it to CPS.

```
> (define list1 (string->list "hello, "))
> list1
(#\h #\e #\l #\l #\o #\, #\space)
> (define list2 (string->list "world"))
> list2
(#\w #\o #\r #\l #\d)
> (list->string (append list1 list2))
"hello, world"
> (append-cps list1 list2 list->string)
"hello, world"
```

Interpreters (25 points)

The abstract syntax of a `while` expression is:

```
public class WhileExp extends AST {
  private AST c, b;
  public WhileExp (AST condition, AST body) { c=condition; b=body; }
  public AST getCondition () { return c; }
  public AST getBody () { return b; }
}
```

The evaluation of the expression proceeds as expected, repeatedly testing the condition, and evaluating the body if the condition is true. Otherwise, the evaluation of the `while` expression terminates and returns 0.

Write the part of the interpreter that evaluates a `while` expression. Make any reasonable assumptions about the rest of the interpreter but document them.

Higher-Order Functions and Closures (25 points)

In a Scheme interpreter, the evaluation of a lambda-expression produces a closure that contains the formal parameters, the body of the function, and the lexical environment. Since the only thing one can do with a closure is to apply it, it could be represented as an object with one method called `apply` that takes the arguments and returns the result. Use this idea to complete the translation of the following Scheme program to Java.

Scheme code:

```
(define cons (lambda (a b) (lambda (s) (s a b))))
(define car (lambda (l) (l (lambda (a b) a))))
(define cdr (lambda (l) (l (lambda (a b) b))))
```

Java code:

```
class Primitives {
  static PairClosure cons (Object a, Object b) { return new PairClosure(a,b); }
  static Object car (PairClosure l) { return l.apply(new SelectFirst()); }
  static Object cdr (PairClosure l) { return l.apply(new SelectSecond()); }
}

class PairClosure {









}









class SelectFirst extends Selector {


}

class SelectSecond extends Selector {


}
```

Continuation-Passing Style (30 points)

Convert the following collection of Scheme functions to continuation-passing style.

```
(define zeroEncoding
  (lambda (f x)
    x))

(define oneEncoding
  (lambda (f x)
    (f x)))

(define twoEncoding
  (lambda (f x)
    (f (f x))))

(define succEncoding
  (lambda (m)
    (lambda (f x)
      (f (m f x)))))

(define addEncoding
  (lambda (m n)
    (lambda (f x)
      (m f (n f x)))))

(define test
  (lambda (nEncoding)
    (let ((n (addEncoding (succEncoding twoEncoding) nEncoding)))
      (n add1 0))))
```

Static vs Dynamic Typing (12 points)

Safe languages like Java and Scheme ensure the integrity of their data. For example, in both languages it is impossible to multiply two strings. Java has a static type checker that detects such situations at compile time. As the interpreter we developed in class illustrates, Scheme detects such situations at run time.

- Show with an example that Java's type checker rejects some expressions that would run in Scheme without any errors.

- Argue that Java *must* reject some expressions that would run in Scheme without any errors. In other words, it is impossible to statically determine, at compile time, if the evaluation of an expression will produce a type error.

Extra Credit (10 points)

The semantics of function application is explained using the β-rule of the λ-calculus:

$$((\lambda x.e)\ e') = e[x := e']$$

where $e[x := e']$ denotes the substitution of all occurrences of x in e by e'. For example:

$$((\lambda x.x + x)3) = (x + x)[x := 3] = 3 + 3 = 6$$

In a formal sense, the above definition is not quite right. But you should have an intuitive understanding of function application and that's what this problem is testing.

Using the β-rule, evaluate the following expressions:

- $(((\lambda f.f)\ (\lambda x.x))\ 3)$

- $((\lambda x.(x\ (\lambda x.x + x)))\ (\lambda f.(f\ 3)))$

- $((\lambda y.(((\lambda x.(\lambda y.x + y))\ y)\ 4))\ 3)$